

# The Arcade Learning Environment: An Evaluation Platform for General Agents

**Marc G. Bellemare**

*University of Alberta, Edmonton, Alberta, Canada*

MG17@CS.UALBERTA.CA

**Yavar Naddaf**

*Empirical Results Inc., Vancouver, British Columbia, Canada*

YAVAR@EMPIRICALRESULTS.CA

**Joel Veness**

VENESS@CS.UALBERTA.CA

**Michael Bowling**

*University of Alberta, Edmonton, Alberta, Canada*

BOWLING@CS.UALBERTA.CA

## Abstract

In this article we introduce the Arcade Learning Environment (ALE): both a challenge problem and a platform and methodology for evaluating the development of general, domain-independent AI technology. ALE provides an interface to hundreds of Atari 2600 game environments, each one different, interesting, and designed to be a challenge for human players. ALE presents significant research challenges for reinforcement learning, model learning, model-based planning, imitation learning, transfer learning, and intrinsic motivation. Most importantly, it provides a rigorous testbed for evaluating and comparing approaches to these problems. We illustrate the promise of ALE by developing and benchmarking domain-independent agents designed using well-established AI techniques for both reinforcement learning and planning. In doing so, we also propose a methodology for evaluation made possible by ALE, reporting empirical results on over 55 different games. All of the software, including the benchmark agents, is publicly available.

## 1. Introduction

A goal of artificial intelligence has long been the development of algorithms capable of general competency in a variety of tasks and domains without the need for domain-specific tailoring. Empirically evaluating general competency is not easily done. Ideally, you would want to compare the technique across domains, which are (i) *varied* enough to claim generality, (ii) each *interesting* enough to be representative of settings that might be faced in practice, and (iii) each created by an *independent* party to be free of experimenter's bias. Typical evaluation in reinforcement learning and planning, though, consists of either just a handful of benchmark domains, or many domains randomly sampled from either an uninteresting distribution (e.g., sampling a random MDP) or a set of domains that is far from varied (e.g., altering the parameters of a standard benchmark domain). Furthermore, it remains common for an algorithm to only be evaluated after considerable domain-specific tailoring, e.g., in specifying a function approximation scheme for the domain, or tuning an algorithm's parameters. In summary, it is unclear how well current experimental methodologies are accurately assessing a technique's general competency. In this article, we introduce the Arcade Learning Environment (ALE): a new challenge problem, platform, and experimental methodology for empirically assessing general artificial intelligence technologies.

ALE is a software framework for interfacing with emulated Atari 2600 game environments. The Atari 2600 is a second generation game console originally released in 1977 which was massively popular for over a decade. Over 500 games were developed for the Atari 2600, spanning a diverse range of genres such as shooters, beat’em ups, puzzle, sports, and action-adventure games; many game genres being first pioneered on the console. While modern game consoles involve visuals, controls, and a general complexity that rivals the real world, Atari 2600 games are far simpler. In spite of this, they still manage to pose a variety of challenging and interesting situations for human players. In summary, the Atari 2600 is a source of *many, varied, interesting, and independently developed* domains, and ALE is a platform for evaluating agent performance across these domains.

ALE is both an experimental methodology and a challenge problem for general AI competency. In machine learning, it is not considered good experimental practice to both train and evaluate an algorithm on the same data set, as it will grossly over-estimate the algorithm’s performance. The typical practice is instead to train on a *training set* with evaluation on a disjoint *test set*. With the large number of available games in ALE, we propose that the same methodology can be used for the same effect: tuning an approach’s parameter values or domain representation on a small number of *training games* and then evaluating on unseen *testing games*. While general competency remains the long-term goal for artificial intelligence, ALE proposes an achievable stepping stone: techniques for general competency across the gamut of Atari 2600 games. We believe this represents a goal that is both attainable in a short time-frame but formidable enough to require new technological breakthroughs.

## 2. Arcade Learning Environment

We begin by describing our main contribution, the Arcade Learning Environment (ALE). ALE is a software framework designed to make it easy to develop and agents that play arbitrary Atari 2600 games.

### 2.1 The Atari 2600

The Atari 2600 is a home video game console developed in 1977, which continued being sold for over a decade Montfort and Bogost (2009). It popularized the the use of general purpose CPUs in game console hardware, with game code distributed through cartridges. Over 500 original games were released for the console, and “homebrew” games continue to be developed now over thirty years later. The games vary widely and include puzzle, sports, action-adventure, boardgames, and many shooter variants. Some of the original games, such as ADVENTURE and PITFALL!, as well as even the console’s joystick itself are iconic symbols of early video games. And nearly all arcade games of the time, such as PAC-MAN and SPACE INVADERS, were ported to the console.

Despite the number and variety of games developed for the Atari 2600, the hardware is relatively simple. It has a 1.19Mhz CPU, and it so it can be emulated much faster than real-time on modern hardware. The console has only 128 bytes (i.e., 1024 bits) of RAM, in addition to the cartridge ROM (typically 2–4kB), which holds the game code. A single game screen is 160-by-210 with a 128-color palette, and the input to the game is only a digital joystick, allowing for 18 possible “actions”: three positions of the joystick for each

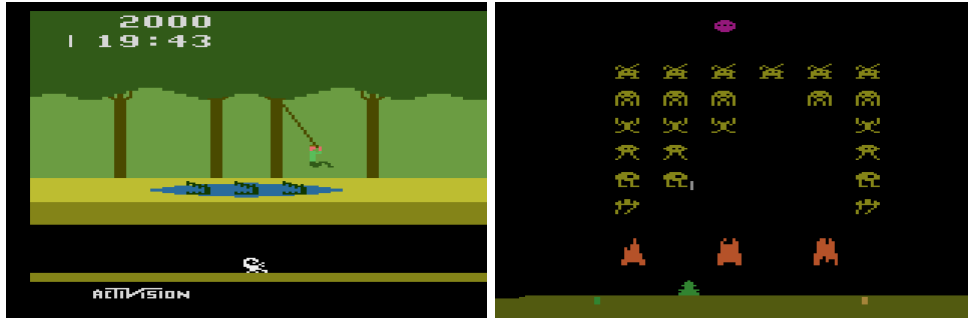


Figure 1: Screenshots of PITFALL! and SPACE INVADERS.

axis, plus a single button. Together, the hardware limits the possible complexity of games, which we believe strikes the perfect balance: challenging but conceivable with near-term advancements in learning, modelling, and planning.

## 2.2 Interface

ALE is built on top of Stella<sup>1</sup>, an open-source Atari 2600 emulator. It allows the user to interface with the Atari 2600, by receiving joystick motions, sending screen and/or RAM information, and emulating the platform. ALE also provides a game-handling layer which identifies, for each game, the accumulated score and when the game ends, transforming it into a standard reinforcement learning problem. In the traditional mode, each observation consists of a single game screen (frame): a 2D array of 7-bit pixels, 160 pixels wide by 210 pixels high. The action space consists of the 18 discrete actions. The game-handling layer can also specify that some actions are unused in a particular game, although none of the results in this paper make use of this information. When running in real-time, the simulator generates 60 frames per second, and at full speed it runs up to 6000 frames per second, although this varies between games. The reward at each time-step is defined on a game by game basis, typically by taking a difference in score or points between frames. An episode begins on the first frame after a reset command is issued, and terminates when the game ends. The game-handling layer also offers the ability to end the episode after a predefined number of frames.<sup>2</sup> The user therefore has access to several dozen games through a single common interface. And, ALE makes adding a new game relatively straightforward.

ALE further provides the functionality to save and restore the state of the emulator. When issued a *save-state* command, ALE saves all the relevant data about the current game, including the contents of the RAM, registers, and address counters. The *restore-state* command similarly resets the game to a previously saved state. This allows us to use ALE as a generative model to study topics such as planning and model-based reinforcement learning.

1. <http://stella.sourceforge.net/>

2. This functionality is needed for a small number of games to ensure that they always terminate. This prevents situations such as in TENNIS, where a degenerate agent could choose to play indefinitely by refusing to serve.

### 2.3 Source Code

ALE is released as free, open-source software under the terms of the GNU General Public License. The latest version of the source code is publicly available at:

<http://arcadelearningenvironment.org>

The source code for the agents used in the benchmark experiments below is also available on the publication page for this article on the same website.

## 3. Benchmark Results

We provide benchmark results for two different AI problem formulations that can be naturally investigated within the ALE framework: reinforcement learning and planning. Our purpose for presenting these results is two-fold. First, they provide a baseline performance for traditional techniques, establishing a point of comparison with future, more advanced, approaches. Second, in describing these results we illustrate our proposed methodology for doing empirical validation with ALE.

### 3.1 Reinforcement Learning

We begin by providing benchmark results using a traditional technique for model-free reinforcement learning. Note that in the reinforcement learning setting, the agent does not have access to a model of the game dynamics. At each time step, the agent selects an action and receives a reward and an observation, and the agent’s aim is to maximize its accumulated reward. In these experiments, learning was performed using the SARSA( $\lambda$ ) algorithm augmented with linear function approximation, replacing traces, and  $\epsilon$ -greedy exploration. A detailed explanation of these techniques can be found in (Sutton and Barto, 1998).

#### 3.1.1 FEATURE CONSTRUCTION

With this approach, the most important design issue is the choice of features to use with linear function approximation. We ran experiments using five different sets of features. These feature sets are now briefly explained. A complete description is given in Appendix A.

**Basic.** The *Basic* method, derived from BASS (Naddaf, 2010), encodes the presence of colors on the Atari 2600 screen. The background is first removed using a histogram method, described in the work of Naddaf (2010). Each game background is precomputed offline, using 18,000 observations collected from sample trajectories. The sample trajectories are generated by following a human-provided trajectory for a random number of steps and subsequently selecting actions uniformly at random. The screen is then divided into  $16 \times 14$  tiles. Basic generates one binary feature for each of the 128 colors and each of the tiles, giving a total of 28,672 features.

**BASS.** The *BASS* method, introduced by Naddaf (2010), behaves identically to the Basic method, save in two respects. First, BASS augments the Basic feature set with pairwise combinations of its features. Second, BASS uses a smaller, 8-colour encoding to make the number of pairwise combinations a tractable size.

**DISCO.** The *DISCO* method, also introduced by Naddaf (2010), aims to detect objects within the Atari 2600 screen. To do so, it first preprocesses 36,000 observations from sample trajectories generated as in the *Basic* method. DISCO also performs the background subtraction steps as in Basic and BASS. Extracted objects are then labelled into classes. During the actual training, DISCO infers the class label of detected objects and encodes their position and velocity using tile coding (Sutton and Barto, 1998).

**LSH.** The *LSH* method maps raw Atari 2600 screens into a small set of binary features using Locally Sensitive Hashing (Gionis et al., 1999). The screens are mapped using random projections, such that visually similar screens are more likely to generate the same features.

**RAM.** The *RAM* method works on an entirely different observation space than the other four methods. Rather than receiving an Atari 2600 screen as an observation, it directly observes the Atari 2600’s 1024 bits of memory. Each bit of RAM is provided as a binary feature together with the pairwise logical-AND of every pair of bits.

### 3.1.2 EVALUATION METHODOLOGY

We first constructed two sets of games, one for training and the other for testing. We used the training games for parameter tuning as well as design refinements, and the testing games for the final evaluation of our methods. Our training set consisted of five games: ASTERIX, BEAM RIDER, FREEWAY, SEAQUEST and SPACE INVADERS. The parameter search involved finding suitable values for the parameters to the SARSA( $\lambda$ ) algorithm, i.e. the learning rate, exploration rate, discount factor, and the decay rate  $\lambda$ . We also searched the space of feature generation parameters, for example the abstraction level for the BASS agent. The results of our parameter search are summarised in Appendix B. Our testing set was constructed by choosing semi-randomly from the 381 games listed on Wikipedia<sup>3</sup> at the time of writing. Of these games, 123 games have their own Wikipedia page, have a single player mode, are not adult themed or prototypes, and can be emulated in ALE. From this list, 50 games were chosen at random to form the test set.

Evaluation of each method on each game was performed as follows. An *episode* starts on the frame that follows the reset command, and terminates when the end-of-game condition is detected or after 5 minutes of real-time play (18,000 frames), whichever comes first. During an episode, the agent acts every 5 frames, or equivalent 12 times per second of gameplay. A reinforcement learning *trial* consists of 5,000 training episodes, followed by 500 evaluation episodes during which no learning takes place. The agent’s performance is measured as the average score achieved during the evaluation episodes. For each game, we report our methods’ average performance across 30 trials.

For purposes of comparison, we also provide performance measures for three simple baseline agents — 0 *Random*, *Const* and *Perturb* — as well as the performance of a non-expert human player. The Random agent picks a random action on every frame. The Const agent selects a single fixed action throughout an episode; our results reflect the highest score achieved by any single action within each game. The Perturb agent selects a fixed action with probability 0.95 and otherwise acts uniformly randomly; for each game, we report the performance of the best policy of this type. Additionally, we provide Human

3. [http://en.wikipedia.org/wiki/List\\_of\\_Atari\\_2600\\_games](http://en.wikipedia.org/wiki/List_of_Atari_2600_games) (July 12, 2012)

Game	Basic	BASS	DISCO	LSH	RAM	Random	Const	Perturb	Human
ASTERIX	862	860	755	<b>987</b>	943	288	650	338	620
SEAQUEST	579	<b>665</b>	422	509	594	108	160	451	156
BOXING	-3	16	12	10	<b>44</b>	-1	-25	-10	-2
H.E.R.O.	6053	<b>6459</b>	2720	3836	3281	712	0	148	6087
ZAXXON	1392	2069	70	<b>3365</b>	304	0	0	2	820

Table 1: Reinforcement Learning results for selected games.

player results that report the five-episode average score obtained by a beginner (who had never previously played Atari 2600 games) playing selected games.

### 3.1.3 RESULTS

A complete report of our reinforcement learning results is given in Appendix C. Table 1 shows just a small subset of results from two training games and three test games. In 40 games out of 55, learning agents perform better than the baseline agents. In some games, e.g., DOUBLE DUNK, JOURNEY ESCAPE and TENNIS, the no-action baseline policy performs the best by essentially refusing to play, and thus incurring no negative reward. Within the 40 games for which learning occurs, the BASS method generally performs best. DISCO performed particularly poorly compared to the other learning methods. The RAM-based agent, surprisingly, did not, outperform image-based methods, despite building its representation from an exact representation of *state*. It appears the screen image carries structural information that is not easily extracted from the RAM bits.

Our reinforcement learning results show that while some learning progress is already possible in Atari 2600 games, much more work remains to be done. Different methods perform well on different games, and no single method performs well on all games. Some games are particularly challenging. For example, platformers such as MONTEZUMA’S REVENGE seem to require high-level planning far beyond what our current, domain-independent methods provide. And, TENNIS requires fairly elaborate behavior before observing any positive reward, but simple behavior can avoid high negative rewards by not ever serving. Our results also highlight the value of ALE as an experimental methodology. For example, the DISCO approach performs reasonably well on the training set, but suffers a dramatic reduction in performance when applied to unseen games. This suggests the method is less robust than the other methods we studied. After a quick glance at the full table of results in Appendix C, it is clear that summarizing results across such varied domains will need further attention. We explore this issue further in Section 4 where we return to these results.

## 3.2 Planning

The Arcade Learning Environment can naturally be used to study planning techniques by using the emulator itself as a generative model. Initially it may seem that allowing the agent to plan into the future with a perfect model would trivialize the problem. However, this is not the case. The massive state-space makes exhaustive search infeasible. To see this, note that 18 different actions are available at every frame. So at 60 frames per second, looking

ahead one second requires  $18^{60} = 10^{75}$  simulation steps. Furthermore, rewards are often sparsely distributed, which causes significant horizon effects in many search algorithms.

### 3.2.1 SEARCH METHODS

We now provide benchmark ALE results for two traditional search methods. Each method was applied online to select an action at every time step until the game was over.

**Breadth-first Search.** Our first approach builds a search tree in a breadth-first fashion until a node limit is reached. Once the tree is expanded, the node values are updated recursively from the bottom of the tree to the root. The agent then selects the action corresponding to the branch with the highest discounted sum of rewards. Expanding the full search tree requires a large number of simulation steps. For instance, selecting an action every 5 frames and allowing a maximum of 100,000 simulation steps per frame, the agent can only look ahead about a third of a second. In many games, this allows the agent to collect immediate rewards and avoid death, but some horizon effects are inevitable. For example, in SEAQUEST the agent must collect a swimmer and return to the surface before running out of air, which involves planning far beyond one second.

**UCT: Upper Confidence Bounds applied to Trees.** A preferable alternative to exhaustively expanding the tree is to simulate deeper into the more promising branches. To do this, we need to find a balance between expanding the higher-valued branches and spending simulation steps on the lower-valued branches to get a better estimate of their values. The UCT algorithm, developed by Kocsis and Szepesvári (2006), deals with the exploration-exploitation dilemma by treating each node of a search tree as a multi-armed bandit problem. UCT uses a variation of UCB1, a bandit algorithm, to choose which child node to visit next. A common practice is to apply a  $t$ -step random simulation at the end of each leaf node to obtain an estimate from a longer trajectory. By expanding the more valuable branches of the tree and doing a random simulation at the leaf nodes, UCT is known to perform well in many different settings (Browne et al., 2012).

Our UCT implementation was entirely standard, except for one optimization. Few Atari games actually distinguish between all 18 actions at every time step. In BEAM RIDER, for example, the down action does nothing, and pressing the button when a bullet has already been shot has no effect. We exploit this fact as follows: after expanding the children of a node in the search tree, we compare the resulting states. Actions that result in the same state are treated as duplicates and only one of the actions is considered in the search tree. This reduces the branching factor, thus allowing deeper search. At every step, we also reuse the part of our search tree corresponding to the selected action.

### 3.2.2 EXPERIMENTAL SETUP

Once again we designed and tuned our algorithms based on the five training games we used in Section 3.1, and subsequently tested the methods on fifty more testing games. The training games were used to determine the length of the search horizon as well as the constant controlling the amount of exploration at internal nodes of the tree. Each episode was set to last up to 5 minutes of real-time play (18,000 frames), with actions selected every 5 frames, matching our settings in Section 3.1.2. We also used the same choice of discount

Game	Full Tree	UCT	Best Learner	Best Baseline
ASTERIX	2136	<b>290700</b>	987	650
SEAQUEST	288	<b>5132</b>	665	451
BOXING	<b>100</b>	100	44	-1
H.E.R.O.	1324	<b>12860</b>	6459	712
ZAXXON	0	<b>22610</b>	3365	2

Table 2: Results for selected games.

factor for each game as we did in Section 3.1. We ran our algorithms for 10 episodes per game. Complete details of the algorithmic parameters can be found in Appendix B.

### 3.2.3 RESULTS

A complete report of our search results is given in Appendix C. Table 2 shows results on a selected subset of games. For reference purposes, we also include the performance of the best learning agent and the best baseline policy from Table 1. Together, our two search methods performed better than both learning agents and the baseline policies on 49 of 55 games. In most cases, UCT performs significantly better than breadth-first search. Four of the six games for which search methods do not perform best are games where rewards are sparse and require long-term planning. These are FREEWAY, PRIVATE EYE, MONTEZUMA’S REVENGE and VENTURE.

## 4. Evaluation Metrics for General Atari 2600 Agents

Applying algorithms to a large set of games as we did in Sections 3.1 and 3.2 present difficulties when interpreting the results. While the agent’s goal in all games is to maximize its score, scores for two different games cannot be easily compared. Each game uses its own scale for scores, and different game mechanics make some games harder to learn than others. The challenges associated with comparing general agents was also previously highlighted by Whiteson et al. (2011). Although we can always report full performance tables as we did in Appendix C, some more compact summary statistics are also desirable. We now introduce some simple metrics that we found useful to compare agents across a diverse set of domains, such as are test set of Atari 2600 games.

### 4.1 Normalized Scores

Consider the scores  $s_{g,1}$  and  $s_{g,2}$  achieved by two algorithms in game  $g$ . Our goal here is to explore methods that allow us to compare two sets of scores  $S_1 = \{s_{g_1,1}, \dots, s_{g_n,1}\}$  and  $S_2 = \{s_{g_1,2}, \dots, s_{g_n,2}\}$ . The approach we take is to transform  $s_{g,i}$  into a normalized score  $z_{g,i}$  with the aim of comparing normalized scores across games; in the ideal case,  $z_{g,i} = z_{g',i}$  implies that algorithm  $i$  performs as well on game  $g$  as on game  $g'$ . In order to compare algorithms over a set of games, we then aggregate normalized scores for each game and each algorithm. For clarity, we use  $m$  to denote the number of games and  $n$  to denote the number of algorithms.



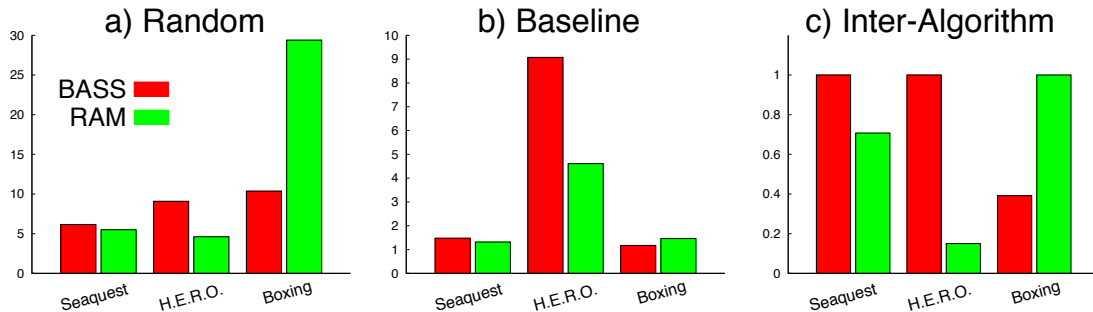


Figure 2: Left to right: random-normalized, baseline and inter-algorithm scores.

The most natural way to compare games with different scoring scales is to normalize scores so that the numerical values become comparable. All of our normalization methods are defined using the notion of a *score range*  $[r_{g,\min}, r_{g,\max}]$  computed for each game. Given such a score range, score  $s_{g,i}$  is normalized by computing  $z_{g,i} := (s_{g,i} - r_{g,\min}) / (r_{g,\max} - r_{g,\min})$ .

#### 4.1.1 NORMALIZATION TO A REFERENCE SCORE

One straightforward method is to normalize to a score range defined by repeated runs of a random agent across each game. Figure 2a depicts the *random-normalized scores* achieved by BASS and RAM on three games. Two issues arise with this approach: the scale of normalized scores may be excessively large and normalized scores are generally not translation invariant. The issue of scale is best seen in a game such as Freeway, for which the random agent achieves a score close to 0: scores achieved by learning agents, in the 10-20 range, are normalized into thousands. In contrast, no learning agent achieves a random-normalized score greater than 1 in Asteroids. We now describe a preferable approach, baseline set normalization.

#### 4.1.2 NORMALIZING TO A BASELINE SET

Rather than normalizing to a single reference we may normalize to the score range implied by a set of references. Let  $b_{g,1}, \dots, b_{g,k}$  be a set of reference scores. A method’s *baseline score* is computed using the score range  $[\min_{i \in \{1, \dots, k\}} b_{g,i}, \max_{i \in \{1, \dots, k\}} b_{g,i}]$ .

Given a sufficiently rich set of reference scores, baseline normalization allows us to reduce the scores for most games to comparable quantities, and lets us know whether meaningful performance was obtained. Figure 2b shows example baseline scores. The score range for these scores corresponds to the scores achieved by 37 baseline policies: the random policy, the 18 constant-action policies and the 18 perturbed-action policies described in Section 3.1.2.

A natural idea is to also include scores achieved by human players into the baseline set. For example, one may include the score achieved by an expert as well as the score achieved by a beginner. Using human scores raises its own set of issues however. For example, humans often play games without seeking to maximize score, and they also benefit from prior knowledge that is difficult to incorporate into domain-independent agents.

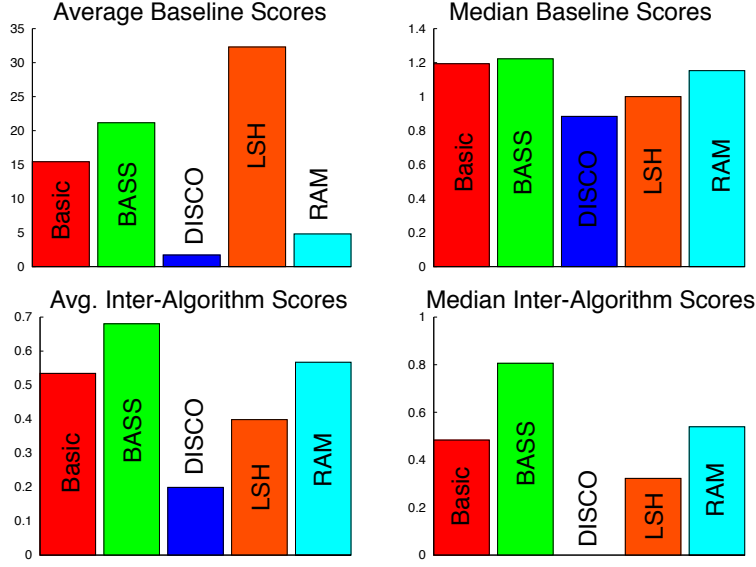


Figure 3: From left to right, top to bottom: average baseline scores, median baseline scores, average inter-algorithm scores and median inter-algorithm scores.

#### 4.1.3 INTER-ALGORITHM NORMALIZATION

A final alternative is to normalize using the scores achieved by the algorithms themselves. Given  $n$  algorithms, each achieving score  $s_{g,i}$  on game  $g$ , we define the *inter-algorithm score* using the score range  $[\min_{i \in \{1, \dots, n\}} s_{g,i}, \max_{i \in \{1, \dots, n\}} s_{g,i}]$ . By definition,  $z_{g,i} \in [0, 1]$ . A special case of this is when  $n=2$ , where  $z_{g,i} \in \{0, 1\}$  indicates which algorithm is better than the other. Figure 2c shows example inter-algorithm scores; the relevant score ranges are constructed from the performance of all five learning agents.

Because inter-algorithm scores are bounded, this type of normalization is an appealing solution to compare the relative performance of different methods. Its main drawback is that it provides no sense of how good the best algorithm is in an objective sense. A good example of this is VENTURE: the inter-algorithm score of 1.0 achieved by BASS does not reflect the fact that none of our agents achieved a score remotely comparable to a human’s performance. The lack of objective reference in inter-algorithm normalization suggests that it should be used to complement other scoring metrics.

## 4.2 Aggregating Scores

Once normalized scores are obtained for each game, the next step is to produce a measure that reflects how well each agent performs across the set of games. As illustrated by Table 3, a large table of numbers does not easily permit comparison between algorithms. We now describe three methods to aggregate normalized scores.

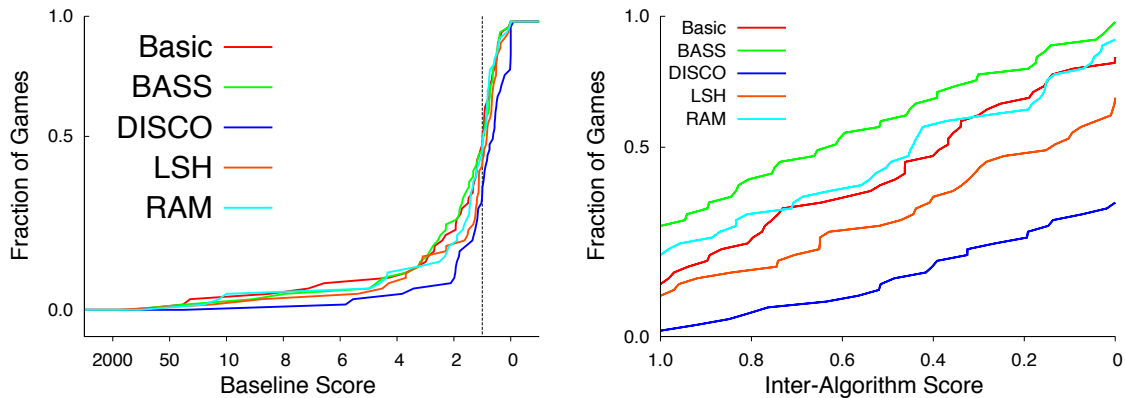


Figure 4: Score distribution over all games.

#### 4.2.1 AVERAGE SCORE

The most straightforward method of aggregating normalized scores is to compute their average. Without perfect score normalization, however, score averages tend to be heavily influenced by games such as ZAXXON for which baseline scores are high. Averaging inter-algorithm scores obviates this issue as all scores are bounded between 0 and 1. Figure 3 displays average baseline and inter-algorithm scores for our learning agents.

#### 4.2.2 MEDIAN SCORE

Median scores are generally more robust to outliers than average scores. The median is obtained by sorting all normalized scores and selecting the middle element (the average of the two middle elements is used if the number of scores is even). Figure 3 shows median baseline and inter-algorithm scores for our learning agents. The median versus average in the baseline score (upper two graphs) illustrates exactly the outlier sensitivity of the average score, where the LSH method appears dramatically superior due entirely to its performance in ZAXXON. However, the median score still paints an incomplete picture.

#### 4.2.3 SCORE DISTRIBUTION

The *score distribution* aggregate is a natural generalization of the median score: it shows the fraction of games on which an algorithm achieves a certain normalized score or better. It is essentially a quantile plot or inverse empirical CDF. Figure 4 shows baseline and inter-algorithm score distributions. Score distributions allow us to compare different algorithms at a glance – if one curve is above another, then it means that the corresponding method generally obtains higher scores.

Using the baseline score distribution, we can easily determine the proportion of games for which methods perform better than the baseline policies (scores above 1). The inter-algorithm score distribution, on the other hand, effectively conveys the relative performance of each method. In particular, it allows us to conclude that BASS performs slightly better than Basic and RAM, and that DISCO performs significantly worse than the other methods.

## 5. Related Work

We now briefly survey some prior work related to Atari 2600 games and on the construction of empirical benchmarks for measuring general competency.

**Atari Games.** There has been some attention devoted to Atari 2600 game playing within the reinforcement learning community. For the most part, prior work has focused on the challenge of finding good state features for this domain. Diuk et al. (2008) applied their DOORMAX algorithm to a restricted version of the game of PITFALL!. Their method extracts objects from the displayed image with game-specific object detection. These objects are then converted into a first-order logic representation of the world, the Object-Oriented Markov Decision Process (OOMDP). Their results show that DOORMAX can discover the optimal behavior for this OOMDP within one episode. Wintermute (2010) proposed a method that also extracts objects from the displayed image and embeds them into a logic-based architecture, SOAR. Their method uses a forward model of the scene to improve the performance of the Q-Learning algorithm (Watkins and Dayan, 1992). They showed that by using such a model, a reinforcement learning agent could learn to play a restricted version of the game of FROGGER. More recently, Cobo et al. (2011) investigated automatic feature discovery in the games of PONG and FROGGER, using their own simulator. Their proposed method takes advantage of human trajectories to identify state features that are important for playing console games. Recently, Hausknecht et al. (2012) proposed HyperNEAT-GGP, an evolutionary approach for finding policies to play Atari 2600 games. Although HyperNEAT-GGP is presented as a general game playing approach, it is currently difficult to assess its general performance as the reported results were limited to only two games. Finally, the authors recently presented a domain independent feature generation technique (Bellemare et al., 2012) that attempts to focus its effort around the location of the player avatar. This work used the evaluation methodology we advocated in earlier sections and is the only to demonstrate the technique across a large set of testing games.

**Evaluation Frameworks for General Agents.** Although the idea of using games to evaluate the performance of agents has a long history in artificial intelligence, it is only more recently that an emphasis on generality has assumed a more prominent role. Pell (1993) advocated the design of agents that, given an abstract description of a game, could automatically play them. His work strongly influenced the design of the now annual General Game Playing competition (Genesereth et al., 2005). Our framework differs in that we do not assume to have access to a compact logical description of the game semantics. Schaul et al. (2011) also presents an interesting recent proposal for using games to measure the general capabilities of an agent. Whiteson et al. (2011) discuss a number of challenges in designing empirical tests to measure general reinforcement learning performance, of which this work can be seen as attempting to address their important concerns.

There have also been a number of attempts to define formal agent performance metrics based on algorithmic information theory. The first such attempts were due to Hernández-Orallo and Minaya-Collado (1998) and Dowe and Hajek (1998). More recently, the approaches of Hernández-Orallo and Dowe (2010) and Legg and Veness (2011) appear to have some potential. Although these techniques are conceptually clean and quite general, the key challenge remains how to specify sufficiently interesting classes of environments for these

frameworks. In our opinion, much more work is required before these approaches can claim to rival the practicality of using a large set of existing human designed environments for agent evaluation.

## 6. Conclusion

This article has introduced the Arcade Learning Environment, a platform for evaluating the development of general, domain-independent agents. ALE provides an interface to hundreds of Atari 2600 game environments, each one different, interesting, and designed to be a challenge for human players. We illustrate the promise of ALE as a challenge problem by benchmarking several domain-independent agents that use well-established reinforcement learning and planning techniques. Our results suggest that general Atari game playing is a challenging, but not intractable problem domain that has the potential to aid the development and evaluation of general agents.

Finally, we conclude by noting that should technology advance so as to render general Atari 2600 game playing achievable, we can always extend our challenge problem to use ever more recent video game platforms. For example, a natural progression might be to next use the Commodore 64, then the Nintendo, Super Nintendo, PlayStation, Xbox, etc. All of these consoles have hundreds of released games with the older platforms having readily available emulators. With the ultra-realism of current generation consoles, each console represents a natural stepping stone toward general real-world competency. Our hope is that by using the methodology advocated in this paper, we can work in a bottom up fashion towards developing more sophisticated AI technology while still maintaining a sufficiently high level of empirical rigor.

## References

- Bellemare, M., Veness, J., and Bowling, M. (2012). Investigating Contingency Awareness Using Atari 2600 Games. In *Proceedings of the the 26th Conference on Artificial Intelligence (AAAI)*.
- Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.
- Cobo, L. C., Zang, P., Isbell, C. L., and Thomaz, A. L. (2011). Automatic state abstraction from demonstration. In *Proceedings of the 22nd Second International Joint Conference on Artificial Intelligence (IJCAI)*.
- Diuk, C., Cohen, A., and Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine learning (ICML)*, pages 240–247.
- Dowe, D. L. and Hajek, A. R. (1998). A non-behavioural, computational extension to the Turing Test. In *Intl. Conf. on Computational Intelligence & multimedia applications (ICCIMA’98), Gippsland, Australia*, pages 101–106.
- Genesereth, M. R., Love, N., and Pell, B. (2005). General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72.

- Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases*, pages 518–529.
- Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. (2012). HyperNEAT-GGP: A HyperNEAT-based Atari general game player. In *Genetic and Evolutionary Computation Conference (GECCO-2012)*.
- Hernández-Orallo, J. and Dowe, D. L. (2010). Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, 174(18):1508 – 1539.
- Hernández-Orallo, J. and Minaya-Collado, N. (1998). A formal definition of intelligence based on an intensional variant of Kolmogorov complexity. In *Proc. Intl Symposium of Engineering of Intelligent Systems (EIS’98)*, pages 146–163. ICSC Press.
- Kanerva, P. (1988). *Sparse distributed memory*. The MIT Press.
- Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293.
- Legg, S. and Veness, J. (2011). An approximation of the universal intelligence measure. *Ray Solomonoff 85th Memorial Conference*.
- Montfort, N. and Bogost, I. (2009). *Racing the Beam: The Atari Video Computer System*. MIT Press.
- Naddaf, Y. (2010). Game-Independent AI Agents for Playing Atari 2600 Console Games. Master’s thesis, University of Alberta.
- Pell, B. (1993). *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, University of Cambridge.
- Schaul, T., Togelius, J., and Schmidhuber, J. (2011). Measuring intelligence through games. *CoRR*, abs/1109.1314.
- Schweitzer, P. J. and Seidmann, A. (1985). Generalized polynomial approximations in markovian decision processes. *Journal of mathematical analysis and applications*, 110(2):568–582.
- Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.
- Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.
- Whiteson, S., Tanner, B., Taylor, M. E., and Stone, P. (2011). Protecting against evaluation overfitting in empirical reinforcement learning. In *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*.
- Wintermute, S. (2010). Using imagery to simplify perceptual abstraction in reinforcement learning agents. In *Proceedings of the the 24th Conference on Artificial Intelligence (AAAI)*.

## Appendix A. Feature Set Construction

This section describes the five feature generation techniques from Section 3.1 in detail.

### A.1 Basic Abstraction of the ScreenShots (BASS)

The idea behind BASS is to directly encode colors present on the screen. This method is motivated by three observations on the Atari 2600 hardware and games:

1. While the Atari 2600 hardware supports a screen resolution of  $160 \times 210$ , game objects are usually larger than a few pixels. Overall, important game events happen at a much lower resolution.
2. Many Atari 2600 games have a static background, with a few important objects moving on the screen. While the screen matrix is densely populated, the actual interesting features on the screen are often sparse.
3. While the hardware can show up to 128 colors in the NTSC mode, it is limited to only 8 colors in the SECAM mode. Consequently, most games use a few number of colors to distinguish important objects on the screen.

The game screen is first preprocessed by subtracting its background, detected using a simple histogram method. BASS then encodes the presence of each of the eight SECAM palette colors at a low resolution, as depicted in Figure 5. Intuitively, BASS seeks to capture the presence of objects of certain colors at different screen locations. BASS also encodes relations between objects by constructing all pairwise combinations of its encoded color features. In ASTERIX, for example, it is important to know if there is a green object (player character) *and* a red object (collectable object) in its vicinity. Pairwise features allow us to capture such object relations.

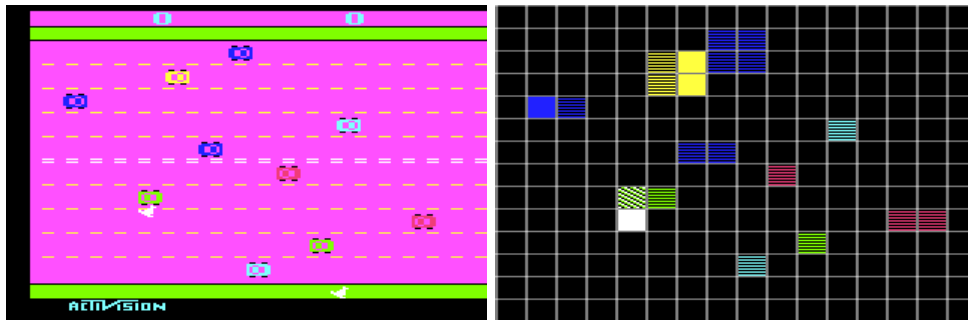


Figure 5: **Left:** Freeway in SECAM colors. **Right:** BASS color encoding for the same screen.

### A.2 Basic BASS

The Basic BASS method generates the same set of features as BASS, but omits the pairwise combinations. This allows us to study whether the additional features are beneficial or

harmful to learning. Because the Basic method has fewer features than BASS, it encodes the presence of each of the 128 colors. In comparison to BASS, Basic therefore represents color more accurately, but cannot represent object interactions.

### A.3 Detecting Instances of Classes of Objects (DISCO)

This feature generation method is based on detecting a set of classes representing game entities and locating instances of these classes on the screen. DISCO is motivated by the following additional observations on Atari 2600 games:

1. The game entities are often instances of a few *classes* of objects. For instance, as Figure 6 shows, while there are many objects in a sample screen of the game FREEWAY, all of these objects are instances of only two classes: *Chicken* and *Car*. Similarly, all the objects on a sample screen of the game SEAQUEST are instances of one of these six classes: *Fish*, *Swimmer*, *Player Submarine*, *Enemy Submarine*, *Player Bullet*, and *Enemy Bullet*.
2. The interaction between two objects can often be generalized to all instances of their respective classes. As an example, consider *Car-Chicken* object interactions in FREEWAY: learning that there is lower value associated with one *Chicken* instance hitting a *Car* instance can be generalized to all instances of those two classes.

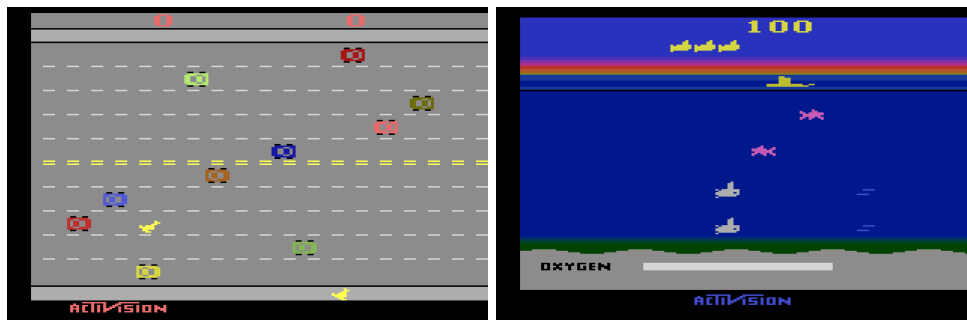


Figure 6: **Left:** Screenshot of the game FREEWAY. Although there are ten different cars, they can all be considered as instances of a single class. **Right:** Screenshot of the game SEAQUEST depicting four different object classes.

DISCO first performs a series of preprocessing steps to discover classes, during which no value function learning is performed. When the agent subsequently learns to play the game, DISCO generates features by detecting objects on the screen and classifying them. The DISCO process is summarized by the following steps:

- Preprocessing:
  - **Background detection:** The static background matrix is extracted using a histogram method, as with BASS.



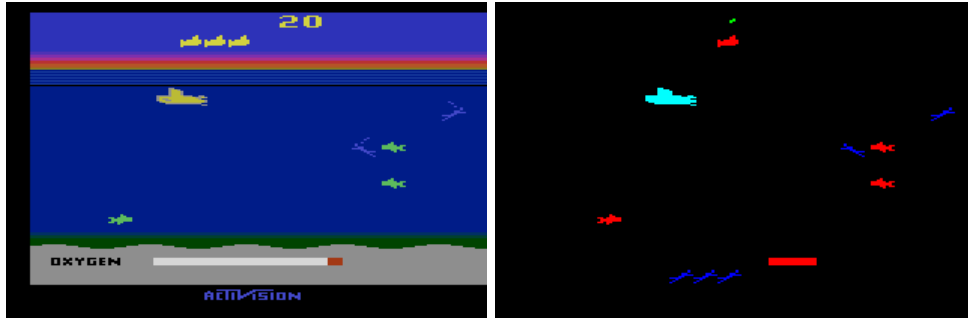


Figure 7: **Left:** Screenshot of the game SEAQUEST. **Right:** Objects detected by DISCO in the game Seaquest. Each color represents a different class.

- **Blob extraction:** A list of moving blob (foreground) objects is detected in each game screen.
- **Class discovery:** A set of classes is detected from the extracted blob objects.
- **Class filtering:** Classes that appear infrequently or are restricted to small region of the screen are removed from the set.
- **Class merging:** Classes that have similar shapes are merged together.
- **Feature generation:**
  - **Class instance detection:** At each time step, class instances are detected from the current screen matrix.
  - **Feature vector generation:** A feature vector is generated from the detected instances by tile-coding their absolute position as well as the relative position and velocity of every pair of instances from different classes.

Figure 7 shows discovered objects in a Seaquest frame. This image illustrates the difficulties in detecting objects: although DISCO correctly classifies the different fish as part of the same class, it also detects a life icon and the oxygen bar as part of that class.

#### A.4 Locality Sensitive Hashing (LSH)

An alternative approach to BASS and DISCO is to use well-established feature generation methods that are agnostic about the type of input they receive. Such methods include polynomial bases (Schweitzer and Seidmann, 1985), sparse distributed memories (Kanerva, 1988) and locality sensitive hashing (LSH) (Gionis et al., 1999). In this paper we consider the latter as a simple mean of reducing the large image space to a smaller, manageable set of features. The input — here, a game screen — is first mapped to a potentially very large bit vector. The resulting vector is then mapped down into a smaller set of features using hashing. LSH performs an additional random projection step to ensure that similar screens are more likely to be binned together. The LSH generation method is summarized as follows:

- Preprocessing:
  - Generate  $l$  random bit vectors, each with  $k$  non-zero entries;
  - Generate one hash function per random bit vector.
- Feature generation:
  - Generate the screen bit vector  $s$ ;
  - Project  $s$  onto each random bit vector;
  - Hash the resulting projection to one of  $M \ll |s|$  features;
  - Concatenate the hashed projections (there are  $l$  of them) into a single feature vector.

### A.5 RAM-based Feature Generation

Unlike the previous three methods, which generate feature vectors based on the game screen, the RAM-based feature generation method relies on the contents of the console memory. The Atari 2600 has only  $128 \times 8 = 1024$  bits of random access memory<sup>4</sup>, which must hold the complete internal state of a game: location of game entities, timers, health indicators, etc. The RAM is therefore a relatively compact representation of the game state, and in contrast to the game screen, it is also Markovian. The purpose of our RAM-based agent is to investigate whether features generated from the RAM affect performance differently from features generated from game screens.

The first part of the generated feature vector simply includes the 1024 bits of RAM. Atari 2600 game programmers often used these bits not as individual values, but as part of 4-bit or 8-bit words. Linear function approximation on the individual bits can capture the value of these multi-bit words. We are also interested in the relation between pairs of values in memory. To capture these relations, the logical-AND of all possible pairs of the bits is appended to the feature vector. Note that a linear function on the pairwise *AND*'s can capture products of both 4-bit and 8-bit words. This is because the product of two  $n$ -bit words can be expressed as a weighted sum of the pairwise products of their bits.

---

4. Some games provided more RAM on the game cartridge: the *Atari Super Chip*, for example, offered an additional 128 bytes of memory. The current approach only considers the main memory included in the Atari 2600 console.

## Appendix B. Experimental Parameters

<b>General</b>	All experiments	Maximum frames per episode	18,000
		Frames per action	5
	Reinforcement learning	Training episodes per trial	5,000
		Evaluation episodes per trial	500
		Number of trials per result	30
<b>Preprocessing</b>	Background detection	Sample screens per game	18,000
	Class discovery	Sample screens per game	36,000
		Maximum number of classes	10
		Maximum object velocity (pixels)	8
		Minimum frequency of class appearance	20%
<b>Reinforcement learning</b>	All agents	Discount factor $\gamma$	0.999
		Exploration rate $\epsilon$	0.05
	BASS and Basic BASS	Learning rate $\alpha$	0.5
		Eligibility traces decay rate $\lambda$	0.9
		Grid width	16
		Grid height	14
	BASS only	Number of different colors	8
	Basic BASS only	Number of different colors	128
	DISCO	Learning rate $\alpha$	0.1
		Eligibility traces decay rate $\lambda$	0.9
		Tile coding, number of tilings	8
		Tile coding, grid size	8
	RAM-based	Learning rate $\alpha$	0.2
		Eligibility traces decay rate $\lambda$	0.5
	LSH	Learning rate $\alpha$	0.5
		Eligibility traces decay rate $\lambda$	0.5
		Number of random vectors $l$	2000
		Number of non-zero vector entries $k$	1000
		Per-vector hash table size $M$	50
<b>Planning</b>	UCT	Simulations per action	500
		Maximum search depth (frames)	300
		Exploration constant	0.1
	Full-tree search	Maximum frames emulated per action	133,000

## Appendix C. Detailed Results

### C.1 Reinforcement Learning

Game	Basic	BASS	DISCO	LSH	RAM	Random	Const	Perturb
ASTERIX	862.3	859.8	754.6	<b>987.3</b>	943.0	288.1	650.0	337.8
BEAM RIDER	929.4	872.7	563.0	793.6	729.8	434.7	<b>996.0</b>	754.8
FREEWAY	11.3	16.4	12.8	15.4	19.1	0.0	21.0	<b>22.5</b>
SEAQUEST	579.0	<b>664.8</b>	421.9	508.5	593.7	107.9	160.0	451.1
SPACE INVADERS	203.6	250.1	239.1	222.2	226.5	156.1	245.0	<b>270.5</b>
ALIEN	<b>939.2</b>	893.4	623.6	510.2	726.4	102.0	140.0	313.9
AMIDAR	64.9	<b>103.4</b>	67.9	45.1	71.4	0.8	31.0	37.8
ASSAULT	465.8	378.4	371.7	<b>628.0</b>	383.6	334.3	357.0	497.8
ASTEROIDS	829.7	800.3	744.5	590.7	907.3	<b>1526.7</b>	140.0	539.9
ATLANTIS	<b>62687.0</b>	25375.0	20857.3	17593.9	19932.7	33058.4	1500.0	12089.1
BANK HEIST	98.8	71.1	51.4	64.6	<b>190.8</b>	15.0	0.0	13.5
BATTLE ZONE	15534.3	12750.8	0.0	14548.1	<b>15819.7</b>	2920.0	13000.0	5772.0
BERZERK	329.2	491.3	329.0	441.0	501.3	233.8	<b>670.0</b>	552.9
BOWLING	28.5	<b>43.9</b>	35.2	26.1	29.3	24.6	30.0	30.0
BOXING	-2.8	15.5	12.4	10.5	<b>44.0</b>	-1.5	-25.0	-10.1
BREAKOUT	3.3	<b>5.2</b>	3.9	2.5	4.0	1.5	3.0	2.9
CARNIVAL	<b>2323.9</b>	1574.2	1646.3	1147.2	765.4	869.2	0.0	485.4
CENTPEDE	7725.5	8803.8	6210.6	6161.6	7555.4	2805.1	<b>16527.0</b>	8937.2
CHOPPER COMMAND	1191.4	<b>1581.5</b>	1349.0	943.0	1397.8	698.2	1000.0	973.7
CRAZY CLIMBER	6303.1	7455.6	4552.9	20453.7	<b>23410.6</b>	2335.4	0.0	2235.0
DEMON ATTACK	520.5	318.5	208.8	355.8	324.8	289.3	130.0	<b>776.2</b>
DOUBLE DUNK	-15.8	-13.1	-23.2	-21.6	-20.3	-15.6	<b>0.0</b>	-20.3
ELEVATOR ACTION	3025.2	2377.6	4.6	<b>3220.6</b>	507.9	1040.9	0.0	562.9
ENDURO	111.8	<b>129.1</b>	0.0	95.8	112.3	0.0	9.0	25.9
FISHING DERBY	-92.6	-92.1	<b>-89.5</b>	-93.2	-91.6	-93.8	-99.0	-97.2
FROSTBITE	161.0	161.1	176.6	<b>216.9</b>	147.9	70.3	160.0	175.2
GOPHER	545.8	<b>1288.3</b>	295.7	941.8	722.5	243.7	0.0	286.8
GRAVITAR	185.3	251.1	197.4	105.9	<b>387.7</b>	205.4	0.0	106.0
H.E.R.O.	6053.1	<b>6458.8</b>	2719.8	3835.8	3281.1	712.0	0.0	147.5
ICE HOCKEY	-13.9	-14.8	-18.9	-15.1	-9.5	-14.8	<b>-1.0</b>	-6.5
JAMES BOND	197.3	<b>202.8</b>	17.3	77.1	133.8	23.3	0.0	82.0
JOURNEY ESCAPE	-8441.0	-14730.7	-9392.2	-13898.9	-8713.5	-18201.7	<b>0.0</b>	-10693.9
KANGAROO	962.4	<b>1622.1</b>	457.9	256.4	481.7	44.4	200.0	498.4
KRULL	2823.3	<b>3371.5</b>	2350.9	2798.1	2901.3	1880.1	0.0	1690.1
KUNG-FU MASTER	16416.2	<b>19544.0</b>	3207.0	8715.6	10361.1	488.2	0.0	578.4
MONTEZUMA'S REVENGE	<b>10.7</b>	0.1	0.0	0.1	0.3	0.3	0.0	0.0
MS. PAC-MAN	1537.2	<b>1691.8</b>	999.6	1070.8	1021.1	163.3	210.0	505.5
NAME THIS GAME	1818.9	2386.8	1951.0	2029.8	2500.1	2012.3	<b>3080.0</b>	1854.3
POOYAN	800.3	1018.9	402.7	<b>1225.3</b>	1210.9	501.1	30.0	540.8
PONG	-19.2	<b>-19.0</b>	-19.6	-19.9	-19.9	-20.9	-21.0	-20.8
PRIVATE EYE	81.9	100.7	-23.0	684.3	111.9	-754.0	0.0	<b>1947.3</b>
Q*BERT	<b>613.5</b>	497.2	326.3	529.1	565.8	169.0	150.0	157.4
RIVER RAID	1708.9	1438.0	0.0	<b>1904.3</b>	1309.9	1608.6	1070.0	1455.5
ROAD RUNNER	67.7	65.2	21.4	42.0	41.0	36.2	<b>900.0</b>	857.9
ROBOTANK	12.8	10.1	9.3	10.8	<b>28.7</b>	1.6	17.0	11.3
SKIING	-1.1	-0.7	-0.1	<b>-0.0</b>	0.0	0.0	0.0	0.0
STAR GUNNER	850.2	<b>1069.5</b>	1002.2	722.9	769.3	638.1	600.0	509.8
TENNIS	-0.2	-0.1	-0.1	-0.1	-0.1	-24.0	<b>0.0</b>	-0.3
TIME PILOT	1728.2	2299.5	0.0	2429.2	<b>3741.2</b>	3458.8	500.0	718.7
TUTANKHAM	40.7	52.6	0.0	85.2	<b>114.3</b>	23.1	0.0	17.3
UP AND DOWN	<b>3532.7</b>	3351.0	2473.4	2475.1	3412.6	131.6	550.0	2962.9
VENTURE	0.0	<b>66.0</b>	0.0	0.0	0.0	0.0	0.0	0.0
VIDEO PINBALL	15046.8	12574.2	10779.5	9813.9	16871.3	<b>20021.1</b>	705.0	9527.9
WIZARD OF WOR	1768.8	<b>1981.3</b>	935.6	945.5	1096.2	772.4	300.0	470.3
ZAXXON	1392.0	2069.1	69.8	<b>3365.1</b>	304.3	0.0	0.0	2.0
Times Best	6	17	1	8	8	2	9	4

Table 3: Reinforcement Learning results. See Section 3.1 for details.

## C.2 Planning

Game	Full Tree	UCT	Best Learner	Best Baseline
ASTERIX	2135.7	<b>290700.0</b>	987.3	650.0
BEAM RIDER	693.5	<b>6624.6</b>	929.4	996.0
FREEWAY	0.0	0.4	19.1	<b>22.5</b>
SEAQUEST	288.0	<b>5132.4</b>	664.8	451.1
SPACE INVADERS	112.2	<b>2718.0</b>	250.1	270.5
ALIEN	784.0	<b>7785.0</b>	939.2	313.9
AMIDAR	5.2	<b>180.3</b>	103.4	37.8
ASSAULT	413.7	<b>1512.2</b>	628.0	497.8
ASTEROIDS	3127.4	<b>4660.6</b>	907.3	1526.7
ATLANTIS	30460.0	<b>193858.0</b>	62687.0	33058.4
BANK HEIST	21.5	<b>497.8</b>	190.8	15.0
BATTLE ZONE	6312.5	<b>70333.3</b>	15819.7	13000.0
BERZERK	195.0	553.5	501.3	<b>670.0</b>
BOWLING	25.5	25.1	<b>43.9</b>	30.0
BOXING	<b>100.0</b>	100.0	44.0	-1.5
BREAKOUT	1.1	<b>364.4</b>	5.2	3.0
CARNIVAL	950.0	<b>5132.0</b>	2323.9	869.2
CENTIPEDE	<b>125123.0</b>	110422.0	8803.8	16527.0
CHOPPER COMMAND	1827.3	<b>34018.8</b>	1581.5	1000.0
CRAZY CLIMBER	37110.0	<b>98172.2</b>	23410.6	2335.4
DEMON ATTACK	442.6	<b>28158.8</b>	520.5	776.2
DOUBLE DUNK	-18.5	<b>24.0</b>	-13.1	0.0
ELEVATOR ACTION	730.0	<b>18100.0</b>	3220.6	1040.9
ENDURO	0.6	<b>286.3</b>	129.1	25.9
FISHING DERBY	-91.6	<b>37.8</b>	-89.5	-93.8
FROSTBITE	137.2	<b>270.5</b>	216.9	175.2
GOPHER	1019.0	<b>20560.0</b>	1288.3	286.8
GRAVITAR	395.0	<b>2850.0</b>	387.7	205.4
H.E.R.O.	1323.8	<b>12859.5</b>	6458.8	712.0
ICE HOCKEY	-9.2	<b>39.4</b>	-9.5	-1.0
JAMES BOND	25.0	<b>330.0</b>	202.8	82.0
JOURNEY ESCAPE	1327.3	<b>7683.3</b>	-8441.0	0.0
KANGAROO	90.0	<b>1990.0</b>	1622.1	498.4
KRULL	3089.2	<b>5037.0</b>	3371.5	1880.1
KUNG-FU MASTER	12127.3	<b>48854.5</b>	19544.0	578.4
MONTEZUMA'S REVENGE	0.0	0.0	<b>10.7</b>	0.3
MS. PACMAN	1708.5	<b>22336.0</b>	1691.8	505.5
NAME THIS GAME	5699.0	<b>15410.0</b>	2500.1	3080.0
POOYAN	909.7	<b>17763.4</b>	1225.3	540.8
PONG	-20.7	<b>21.0</b>	-19.0	-20.8
PRIVATE EYE	57.9	100.0	684.3	<b>1947.3</b>
Q*BERT	132.8	<b>17343.4</b>	613.5	169.0
RIVER RAID	2178.5	<b>4449.0</b>	1904.3	1608.6
ROAD RUNNER	245.0	<b>38725.0</b>	67.7	900.0
ROBOTANK	1.5	<b>50.4</b>	28.7	17.0
SKIING	<b>0.0</b>	-0.8	0.0	0.0
STAR GUNNER	<b>1345.0</b>	1207.1	1069.5	638.1
TENNIS	-23.8	<b>2.8</b>	-0.1	0.0
TIME PILOT	4063.6	<b>63854.5</b>	3741.2	3458.8
TUTANKHAM	64.1	<b>225.5</b>	114.3	23.1
UP AND DOWN	746.0	<b>74473.6</b>	3532.7	2962.9
VENTURE	0.0	0.0	<b>66.0</b>	0.0
VIDEO PINBALL	55567.3	<b>254748.0</b>	16871.3	20021.1
WIZARD OF WOR	3309.1	<b>105500.0</b>	1981.3	772.4
ZAXXON	0.0	<b>22610.0</b>	3365.1	2.0
Times Best	4	45	3	3

Table 4: Search results. See Section 3.2 for details.